Abstract

Graph compression, or sparsification, is an effective pre-processing step for many algorithms which are required to operate on large graphs, as graphs of reduced size may be used in their place. Compression is inhibited due to the ineffectiveness of many existing exact compression methods, where compression itself may take several minutes or potentially hours for the largest of graphs. As the sizes of modern graphs continue to grow, so does the need for faster compression methods such as those which provide approximate results. This project aims to develop a linear time approximate graph compression algorithm using random edge contractions and vertex sparsification techniques. The goal of employing such an approach is to provide efficient compression while minimising the negative effects of approximate operations. Several variations of this algorithm were investigated and an early halting variant of the approximate sparsifier was found to provide very good results in terms of quality and efficiency, even for relatively large graphs.

Contents

1 Introduction

Graph compression, or sparsification, is a process by which a graph may be compressed, reducing its size. This is an effective pre-processing step for algorithms which operate on large graphs, as a smaller version of the graph may be used instead with the aim of improving efficiency. This report considers distance preserving sparsifiers (Gupta (2001)).

Definition 1.1 (Distance Preserving Sparsifier) *Given a graph $G = (V, E, c)$, a real value $\alpha \geq 1$, and a set of vertices $T \subseteq V$, a distance preserving sparsifier aims to construct a graph $H = (V', E', c)$ where $V' \supseteq T$ such that $dist_H(u, v) \leq \alpha \cdot dist_G(u, v), \forall u, v \in T$.*

In the above definition α is a worst-case guarantee, and therefore represents the largest value over all pairs of terminals. Vertex sparsification (Moitra (2009), Krauthgamer and Zondiner (2012), Räcke et al. (2014), Kamma et al. (2015), Cheung et al. (2016)) is a compression technique in which certain vertices, known as non-terminals, are systematically removed from a graph. A sparsifier will fall into one of two categories, an exact, or perfect, sparsifier will construct a graph where distances are preserved exactly, or so that the value of $\alpha = 1$, and an approximate sparsifier will result in a value of $\alpha \geq 1$. This project aims to develop and benchmark an approximate vertex sparsifier and to compare it to existing exact sparsifiers to measure the performance of the algorithm. The remainder of this section outlines the motivations for exact and approximate sparsifiers and section 2 covers definitions used throughout this report. Section 3 discusses similar work in the field of sparsification. Section 4 details the analysis and plan of the development of the approximate sparsifier. In section 5 a high level outline of the sparsification techniques used in this project is given, this is furthered in section 6 which discusses the implementations of each algorithm. Section 7 covers the experimental analysis and comparisons between sparsifiers, and finally section 8 summarises this work and outlines several opportunities for further research.

1.1 Motivation

The motivation for graph sparsification is shown by the prevalence of large graphs in many disciplines. Certain graphs such as road network graphs contain millions of vertices, and others, for example social media graphs, may even contain billions. Due to their sheer size, computations on modern graphs can become very time-consuming and computationally expensive. Graph compression allows for compact representations of such large graphs, with the graph of reduced size maintaining important qualities of the original. Graph algorithms may operate on this smaller graph by using compression as a pre-processing step in order to improve efficiency. Large graphs themselves hinder this pre-processing, as compression itself is a time-consuming operation exaggerated by the size of modern graphs. The exact sparsifier based on Gaussian elimination (Tarjan (1975)), discussed in sections 5.1 and 6.2.4, has a worst-case time complexity of $O(m \cdot n^2)$ for $m = |V| - |T|$ and $n = |V|$ with many large graphs reaching a stage of pathological input during compression. Expensive exact sparsifiers and the emergence of huge graphs serve to elucidate the motivation for approximate sparsifiers. In such sparsifiers the process is usually much faster than in exact sparsification and lowers the computational cost significantly, however metrics, such as shortest path distances, may not be preserved well after compression.

2 Preliminaries

An undirected weighted graph, G, is defined as $G = (V, E, c)$ where V represents a set of vertices, or nodes, and E is a set of unordered pairs of vertices called edges, defined as $E = \{(u, v)|u, v \in V \wedge u \neq v\}$. Each edge has a real number assigned to it known as its weight. For each edge $(u, v) \in E$, u and v are referred to as *adjacent*, and each vertex is labelled as *incident* to that edge. For two vertices, u and v, $c(u, v)$ is a function that returns the weight of the edge $(u, v) \in E$ and 0 if $(u, v) \notin E$. $dist_G(u, v)$ is an external function that returns the shortest path between two vertices, u and v, as the sum of the weights of edges that lie on the shortest path between the two vertices.

The algorithms used in this project often consider the *neighbourhood* and *degree* of a vertex, these are related and their definition is as follows:

Definition 2.1 (Neighbourhood and Degree of a Vertex) *Given a graph $G = (V, E, c)$ and some vertex $v \in V$, the neighbourhood of v is a subgraph of G, $N = (V', E', c)$ where $V' = \{u|(u, v) \in E\}$ and $E' = \{(u, v)|(u, v) \in E \wedge u \in V' \wedge u \neq v\}$. The degree of v is defined as $deg(v) = |E'|$, or alternatively the number of edges v is incident to.*

A distance preserving sparsifier aims to construct $H = (V', E', c)$ from some graph $G = (V, E, c)$ where path distances are preserved. Sparsifiers fall into one of two categerories, defined below:

Definition 2.2 (Exact and Approximate Sparsifiers) *Following on from definition 1.1, a distance preserving sparsifier aims to construct a graph where $dist_H(u, v) \leq \alpha \cdot dist_G(u, v), \forall u, v \in T \wedge \alpha \geq 1$. For an exact sparsifier $\alpha = 1$, and an approximate sparsifier will have a value of $\alpha \geq 1$.*

This project considers *vertex sparsification*, a technique where vertices deemed unimportant are systematically removed from a graph. These vertices are known as *non-terminals*, and it follows that vertices deemed important are known as *terminals*. For the graph G, the set of terminals is defined as $T \subseteq V$ and therefore the set of non-terminals as $V - T$.

Definition 2.3 (Vertex Sparsifier) *Following on from definition 1.1, given a graph $G = (V, E, c)$ and a set of terminals $T \subseteq V$, a vertex sparsifier constructs a graph $H = (V', E', c)$ where $V' \supseteq T$. In vertex sparsification this is performed via the systematic removal of vertices in the set $V - T$ from V.*

In definition 1.1, the value α represents a worst-case guarantee in that it is the largest value found over all pairs of terminals. This value may also be referred to as the *quality*, q, or the quality of a sparsifier and represents how well the compressed graph represents the original. Another term for this value specific to distance preserving sparsifiers is *stretch*, as the value of α represents by how much the path length between a pair of terminals has increased during compression.

3 Background

Ideas contributing to graph compression have long existed prior to its formal definition, for example a modification of an early graph theoretic approach to performing Gaussian elimination (Tarjan (1975)) is used in exact graph compression. This approach is detailed in sections 5.1 and 6.2.4. The first formal definition of graph compression constrained the process to reducing the number of edges in a graph efficiently (Feder and Motwani (1995)). Lowering the number of edges in a graph is now recognised as *edge sparsification*, and there are several unique approaches. *Vertex sparsification* is another common compression method, where the number of vertices is lowered instead. Each procedure is not limited to operating on the property the name indicates, and sparsifiers may reduce the number of edges, vertices, or both.

An important property of sparsifiers is that they preserve some metric in order to accurately represent the original graph. *Cut sparsification* (Benczúr and Karger (1996)) is one example in which the aim is to preserve minimum cuts in the compressed graph, and an approach to cut sparsification using random sampling (Karger (1994)) motivated approximate compression algorithms with relative success. A cut defines two subsets of vertices with edges connected by endpoints in each set. Spectral sparsifiers (Spielman and Srivastava (2008), Spielman and Teng (2011)) are another approach in which the aim is to approximately preserve the Laplacian after compression. These definitions all fall into the category of edge sparsification. The preservation of similar metrics, such as cut sparsification (Moitra (2009)) and flow sparsification (Räcke et al. (2014)) also appear frequently in vertex sparsification. Flow sparsification aims to preserve network flow after compression.

Distance preserving sparsifiers (Gupta (2001)) were first introduced for the compression of trees and aim to preserve distances between terminals in the compressed graph. The utility of preserving distances is apparent for operations where this quality is essential, such as in routing using compression hierarchies (Geisberger et al. (2008)) and other tasks such as graph-based optimisation problems. Some popular approaches to distance preserving sparsifiers are based on constructing graph minors, subgraphs created through the deletion of vertices, edges, or both from a graph. These approaches can either be approximate (Cheung et al. (2016)) or exact (Krauthgamer and Zondiner (2012), Kamma et al. (2015)). Exact approaches prove effective as minors may be created directly using shortest path algorithms, and this process is discussed in sections 5.2 and 6.2.5.

This work contributes what is believed to be a novel sparsification approach which combines several techniques employed by the aforementioned sparsifiers. Edge contractions appear in many sparsifiers for various applications (Benczúr and Karger (1996), Karger (1994), Spielman and Teng (2011), Geisberger et al. (2008), Cheung et al. (2016), Krauthgamer and Zondiner (2012), Kamma et al. (2015)) and these are to be combined with random sampling approaches justified in prior sparsifiers (Karger (1994), Benczúr and Karger (1996)). The minimum degree heuristic is also a commonality in many graph and matrix algorithms (Rose (1970), Tarjan (1975)), as are operations on vertex neighbourhoods. A more in-depth description of this project's contribution is given in the following sections.

4 Problem Analysis

This section discusses and justifies the design choices made when developing the new approximate sparsifier. More in-depth details of the functionality of the algorithm are given in section 5.3, and section 5 overall gives a high-level description of all sparsifiers used in this work. Implementation specifics for the approximate sparsifier and all other algorithms and data structures are given in section 6. Java was chosen for implementations primarily due to familiarity with the language. The only constraint of the approximate sparsifier is that it contracts edges randomly in order to remove non-terminals from a graph, other than this it is unquestionable that the algorithm should be as efficient as possible. The algorithm will be designed as a method where, given a graph $G = (V, E, c)$, and a non-terminal $v \in V$, v will be removed from V, non-terminals may then be passed to the algorithm individually as needed, and this encourages the development of hybrid sparsifiers which combine multiple sparsification techniques.

4.1 Minimum Degree Heuristic

The minimum degree heuristic is widely used in many graph and matrix algorithms, and has been referred to in works dating back to the 1970s (Rose (1970), Tarjan (1975)). Using this approach vertices will be processed in order of their degree, with the vertex of minimum degree being prioritised. Due to widespread usage it was decided that this heuristic would be employed in order to provide sparsifiers with non-terminals to contract. As edge contractions will impact edges that a non-terminal is incident to, this may also aid in preserving quality as fewer edges will be affected at each compression step. The use of this heuristic also justifies the use of a min-priority queue, as this data structure allows for the retrieval of the non-terminal with minimum degree in $O(1)$ and the population of the queue in $O(n)$, or $O(\log n)$ where a heap structure is used. Dijkstra's shortest path algorithm (Dijkstra (1959)), used for quality calculations as discussed in sections 4.4 and 6.2.3, may also be implemented efficiently using a min-priority queue and therefore the utility of such a structure is clear.

4.2 Random Sampling and the Alias Method

As mentioned previously the sparsifier should select edges to contract randomly. Random sampling has been employed and justified as a suitable approach in previously developed sparsifiers and other graph and matrix algorithms (Karger (1994), Karger (1999), Karger (2000), Kyng and Sachdeva (2016)). This process should be as efficient as possible, and therefore the alias sampling method (Kronmal and Peterson Jr (1979)) will be used. This method is discussed in detail in sections 5.3 and 6.1.2, however a brief description is given here to motivate the choice. The alias method allows the sampling of a discrete distribution as though it were a uniform distribution through the partitioning of item probabilities, and the process is separated into two distinct stages, partitioning and sampling. In the partitioning stage, n items are partitioned based on their probabilities and this may be represented as an indexable list of partitions created in $O(n)$.

During the sampling stage, a pseudo-random uniform number may be generated incredibly efficiently in $O(1)$, and using this number the task is then to simply index the partitions in order to retrieve an item, overall the sampling process may be carried out in constant time, and therefore the alias sampling method has an overall time complexity of $O(n)$. The implementation is slightly more complex than this simplified view of the algorithm and is discussed in sections 5.3 and 6.1.2. During the partitioning process, partitions are filled by retrieving the two items with smallest and largest probabilities at a given time. As these values will change throughout the process, a data structure is required for the efficient retrieval of these values. A self-balancing binary tree, specifically an AVL-Tree (AdelsonVelskii and Landis (1963)) was selected for this task as insertion and the retrieval of the minimum and maximum values may be performed in $O(\log n)$ for each operation, as only one path must be traversed within the tree. Before partitioning, the tree is populated with n items in $O(n \cdot \log n)$ and for each partition the items with smallest and largest probabilities are retrieved from the tree in $O(\log n)$, the resulting complexity of the partitioning step then becomes $O(n \cdot \log n + n \cdot (2 \cdot \log n)) = O(n \cdot \log n)$.

4.3 Graph Data Structure

A relatively simple and widely used implementation of a graph data structure was employed, and is described in more detail in section 6.1. The graph data structure itself maintains an indexable list of vertex data structures, each of which indicates its index within the graph and also maintains an adjacency list as a hashmap used to represent edges. The key value of the hashmap is used to represent the index of the adjacent vertex, and the value of the entry represents the weight of the edge between the two vertices. A hashmap was chosen as both insertion and retrieval may be performed in constant time, however this is not guaranteed and retrieval may be $O(n)$ in the worst case where hashes are identical. The decision to implement the graph in this way was motivated by the need for the efficient retrieval of edges during partitioning and edge contractions in the approximate sparsifier. Another small addition to the vertex data structure is to maintain a sum of hashmap values, or edge weights, upon their insertion into the adjacency list. This allows for the calculation of probabilities in constant time during the partitioning stage of the alias method.

4.4 Quality Assessments

Lastly, quality assessments must calculate shortest paths in both the compressed and uncompressed graphs and this process is discussed in detail in section 6.2.3. Quality values will be calculated between all pairs of terminals in the graph, and therefore an all pairs shortest paths computation is required. For a graph $G = (V, E, c)$ with terminal set T, this may be done in two ways, either using the Floyd-Warshall Algorithm (Floyd (1962)) with a time complexity of $O(|V|^3)$, or using Dijkstra's shortest path algorithm (Dijkstra (1959)) starting from each terminal in the graph in $O(|T| \cdot (|V| + |E| \cdot \log |V|))$. The approach using Dijkstra's shortest path algorithm was chosen as both strategies provide identical results with poor run times. Investigation into potentially faster run times could be explored and this is discussed in sections 6.2.3 and 8.

5 Sparsification Techniques

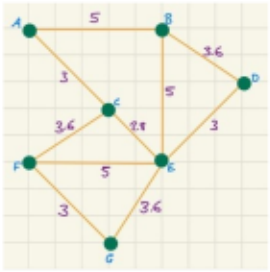
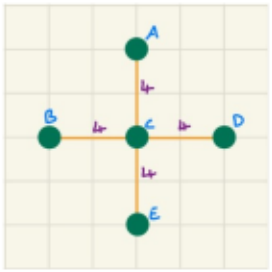
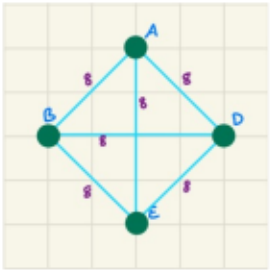


Figure 5.1: *A small weighted graph G consisting of 7 vertices shown in green. Edge weights are shown in purple and vertex labels in blue. The terminal set is defined as* $T = \{A, D, E, F\}$.

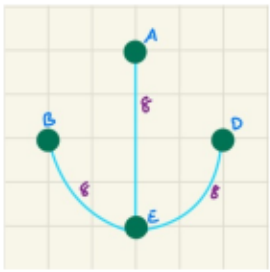
This section provides high level descriptions of all vertex sparsifiers used throughout the project. A more in-depth description of each algorithm with pseudocode and time complexity analysis can be found in section 6. The sparsifiers include two existing exact sparsifiers, the first of which is a modification of a Gaussian elimination algorithm (Tarjan (1975)) and the second using shortest path trees to construct graph minors (Krauthgamer and Zondiner (2012)), and one newly developed approximate sparsifier which makes use of random edge contractions. Examples and visualisations throughout this section are based on the example graph G shown in figure 5.1. An example of the effects of stretch, briefly mentioned in section 2 and governed by the value α in definition 1.1, can be seen in figure 5.2. Figure 5.2b shows the graph compressed using the Gaussian elimination method, it can be seen that a clique is formed on the neighbourhood of the non-terminal to be removed, and therefore shortest paths such as the path between B and D have their distances preserved exactly. Figure 5.2c shows the same graph compressed using REC. Here the same shortest path has doubled and must pass through E. Unlike in the exact compression no clique is formed on the neighbourhood and therefore the path passes through E.



(a) *A small weighted graph S with one non-terminal, C.*



(b) *The graph S compressed using the Gaussian elimination sparsifier discussed in section 5.1.*



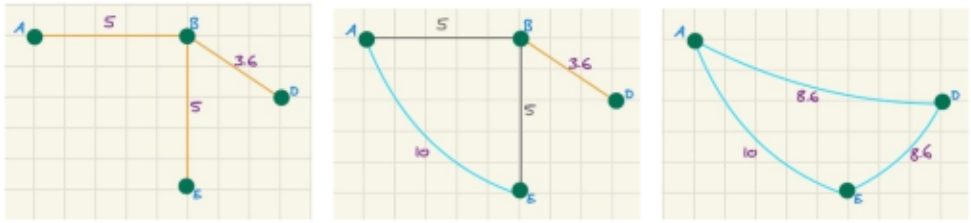
(c) *The graph S compressed using the REC sparsifier discussed in section 5.3.*

Figure 5.2: *An example of the effects of stretch.*

5.1 Gaussian Elimination Method

The Gaussian Elimination method is an exact sparsifier, and is a modification of an algorithm for performing Gaussian elimination on a graph representation of a matrix (Tarjan (1975)). Gaussian elimination is a process through which a system of linear equations can be solved.

This method considers the neighbourhood of each non-terminal. During the elimination of some non-terminal w, every pair of neighbours, u and v, of w are analysed and a new candidate edge (u, v) is created with $c(u, v) = c(u, w) + c(w, v)$. Where the edge (u, v) already exists, its weight is updated to be the lower of the two, and where it does not the candidate edge is added to E. After this process, all edges incident to w are removed from E and w is then removed from V. This process is shown in figure 5.3.



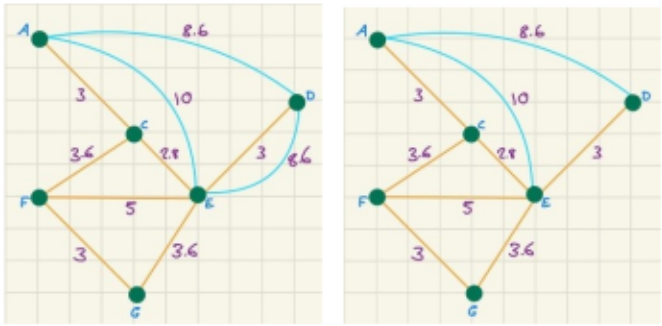
(a) The neighbourhood of B is considered.

(b) A new edge (A, E) (cyan) is created with the weight of the two grey edges.

(c) The process in 5.3b is applied to all pairs of neighbours of B, then B is removed from the graph.

Figure 5.3: *The Gaussian elimination algorithm removing non-terminal B from graph G.*

Figure 5.3 shows a simplified view of the removal process of a vertex in isolation from the complete graph. The result of this process is more accurately represented by figure 5.4a. The process of determining the lower edge weight where the candidate edge is a duplicate is also visualised here, and shows that lower edge weights will be favoured. The algorithm does not account for paths, for example figure 5.4 shows that the edge (A, E) is created with $c(A, E) = 10$, however a path from A to E through C has weight $c(A, C) + c(C, E) = 5.8$. The entire process of final edge selection is shown in figure 5.4.



(a) The graph G with old edges removed after the steps in figure 5.3. Duplicate edges exist between D and E.

(b) The duplicate edge is handled, in this case the old edge has lower weight and so is favoured over the new edge.

Figure 5.4: *The Gaussian elimination algorithm selecting edge weights after the elimination of vertex B.*

Throughout sparsification the graph gradually becomes more connected as each compression step forms a clique on the neighbourhood of the non-terminal to be removed, this is shown in figure 5.3c. Formally, a clique is a set of vertices where all vertices are adjacent to every other vertex in the set. Provided there are enough non-terminals to contract and the graph is large enough, the graph may become fully-connected during sparsification, figure 5.5 shows the graph G after sparsification using this technique and it can be seen that it is almost fully-connected with the omission of a single edge between F and D. The Gaussian elimination method has a worst-case time complexity of $O(m \cdot n^2)$ for the pathological input of a fully-connected graph, where $m = |V| - |T|$ and $n = |V|$ and this is detailed in section 6.2.4. This implies that for a large enough graph, the algorithm may become incredibly slow during compression due to the large amount of edges created by previous compression steps, however it is this quality of the algorithm that causes compression to be exact.

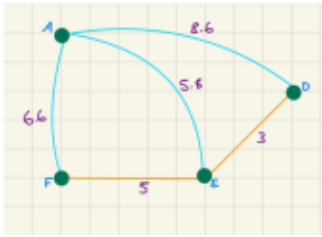


Figure 5.5: *The graph shown in figure 5.1 with all non-terminals removed via the Gaussian elimination method. New edges are shown in cyan and original edges are shown in yellow.*

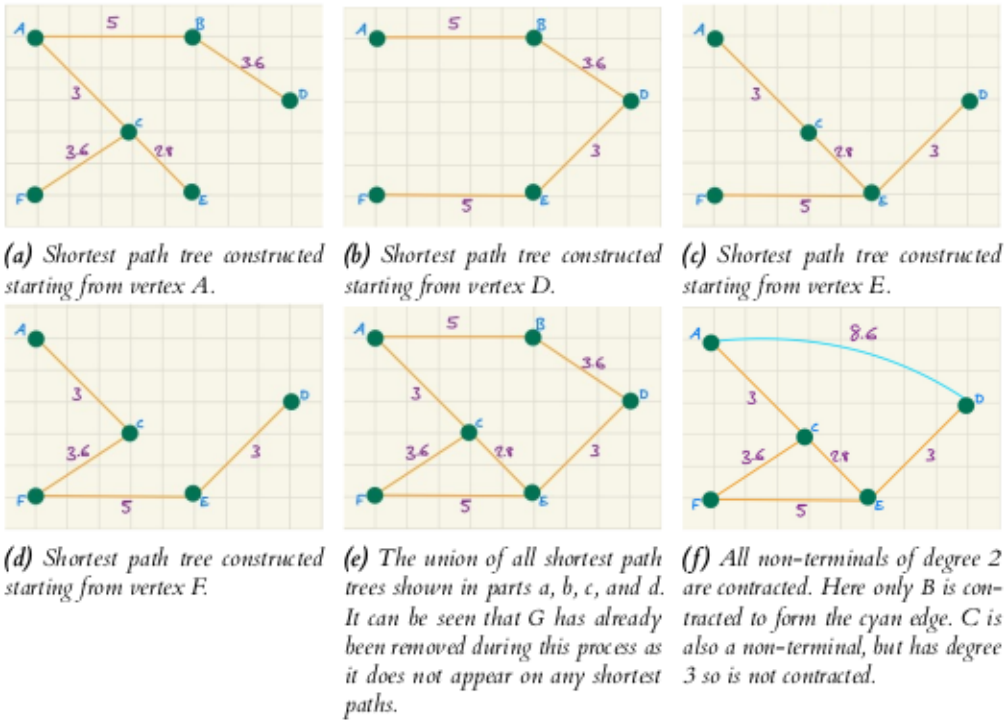
5.2 Shortest Path Tree Method

The Shortest Path Tree method (SPT) (Krauthgamer and Zondiner (2012)) is an approach to sparsification where a distance preserving graph minor is constructed. A graph minor is a subgraph created through the deletion of edges, vertices, or both from a graph. The minor is constructed from *shortest path trees*, a form of *spanning tree* where the distance from the root to any node in the tree is the shortest path between the two vertices in the graph.

> **Definition 5.1 (Spanning Tree and Shortest Path Tree)** *Given a graph $G = (V, E, c)$ and a root node $v \in V$, a spanning tree is a subgraph $S = (V, E', c)$ that contains no cycles. A shortest path tree is a form of spanning tree where the set of edges E' contains only those edges that fall on shortest paths from the root vertex v to all other vertices in V.*

In this implementation, a shortest path tree is constructed using Dijkstra's shortest path algorithm (Dijkstra (1959)). The sparsifier calculates a shortest path tree rooted at each terminal, then a minor is created as the union of all trees. Lower edge weights are favoured where duplicates exist during the union. The resulting minor contains all terminals of the graph and any non-terminals lying on shortest paths between terminals. A form of edge contraction is then carried out on non-terminals in the minor. As terminals are the only vertices that will lie at the end of a path all remaining non-terminals will have degree ≥ 2, a non-terminal whose degree is 2 will be replaced by an edge between its neighbours. This is an example of an exact sparsifier, however it is not a complete sparsification of the graph due to the fact that any non-terminals with degree > 2 will remain in the compressed graph. The process carried out by the algorithm can be seen in figure 5.6. It is evident that this sparsifier is exact as the minor is constructed using shortest path distances, and therefore these will be identical after compression[1]

[1] A proof of this quality is provided in the original paper (Krauthgamer and Zondiner (2012)).



(a) Shortest path tree constructed starting from vertex A.

(b) Shortest path tree constructed starting from vertex D.

(c) Shortest path tree constructed starting from vertex E.

(d) Shortest path tree constructed starting from vertex F.

(e) The union of all shortest path trees shown in parts a, b, c, and d. It can be seen that G has already been removed during this process as it does not appear on any shortest paths.

(f) All non-terminals of degree 2 are contracted. Here only B is contracted to form the cyan edge. C is also a non-terminal, but has degree 3 so is not contracted.

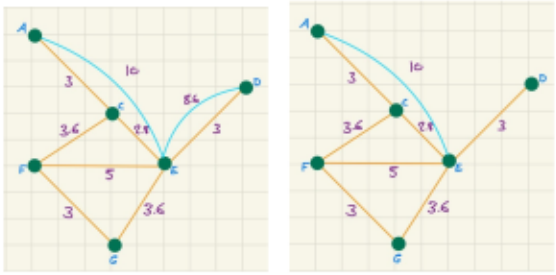
Figure 5.6: *The SPT method constructing a graph minor as the union of all shortest path trees, then contracting all non-terminals of degree 2.*

5.3 Random Edge Contractions Method

The random edge contractions (REC) sparsifier is an approximate sparsifier and the main contribution of this project. The sparsifier aims to compress a graph efficiently by randomly contracting edges. As this approach employs random elements, multiple compressions of the same graph with identical terminals may give different results. The algorithm functions iteratively over the set of non-terminals. At each iteration, the non-terminal with minimum degree is selected for removal from the graph and it is contracted into one of its neighbours, the edge contraction operation is defined as follows:

Definition 5.2 (Edge Contraction) *Given a graph $G = (V, E, c)$ and two adjacent vertices $u, v \in V$, an edge contraction merges the two adjacent vertices u and v into a single vertex w, sometimes referred to as a supernode. u and v are removed from V, the edge (u, v) is removed from E and all edges incident to u and v are updated to be incident to w instead.*

The REC algorithm does not merge two vertices into a supernode, and instead merges a non-terminal with a neighbouring vertex selected via random sampling. Once the two vertices have been merged, duplicate edges are handled in the same manner as in the Gaussian elimination method where the edge with lower weight is favoured. A visualisation of this process is shown in figure 5.7. It can be seen that, similarly to the Gaussian elimination method, the REC method may create an edge between two vertices with a shorter path between them. This figure also serves to show the effects of stretch, as the once shortest path between A and D passing through B had length 8.6, however in the compressed graph the shortest path passes through C and E and has length 8.8.



(a) G with B merged into E by edge contraction. Altered edges are shown in cyan.

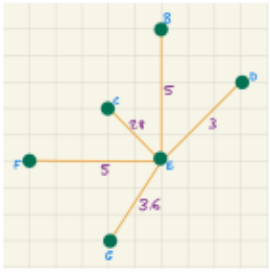
(b) Duplicate edge (D, E) is resolved, here the original edge remains.

Figure 5.7: *The REC algorithm merging two vertices via edge contraction.*

As mentioned previously, the edge to be contracted is selected through random sampling. The alias method for random sampling (Kronmal and Peterson Jr (1979)) is used as an efficient method of sampling from a discrete distribution as though it were a uniform distribution. The sampling process considers the neighbourhood of a vertex, examples will refer to the neighbourhood of vertex E in graph G which is shown in figure 5.8a. Each edge is assigned an *edge weight probability*, for the neighbourhood of E this is shown in table 5.8b.

Definition 5.3 (Edge Weight Probability) *Given the neighbourhood, $N = (V', E', c)$, of some vertex $v \in V'$, the edge weight probability of each edge $(u, v) \in E'$ is defined as*

$$P((u,v)) = \frac{c(u,v)}{\sum_{(w,x)\in E'} c(w,x)}$$



(a) Neighbourhood of vertex E in the graph G (figure 5.1).

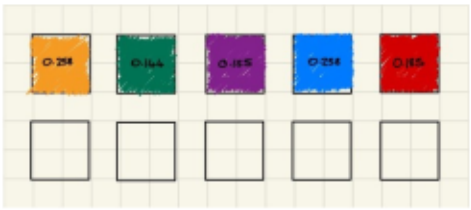
Edge	Probability
$P((E,B))$	5/19.4 = 0.258
$P((E,C))$	2.8/19.4 = 0.144
$P((E,D))$	3/19.4 = 0.155
$P((E,F))$	5/19.4 = 0.258
$P((E,G))$	3.6/19.4 = 0.185

(b) Table of edge weight probabilities for E.

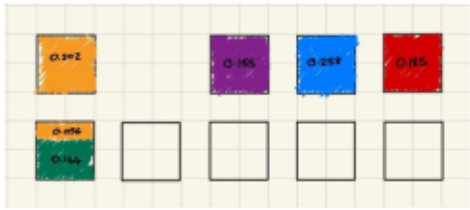
Figure 5.8: *The neighbourhood of vertex E and the corresponding edge weight probabilities.*

The alias method is separated into two distinct stages, partitioning and sampling. In the partitioning stage, n partitions of size $\frac{1}{n}$ are created for n items to sample from. Probabilities are partitioned iteratively, for each empty partition the item with smallest edge weight probability is placed into the partition with its full probability. The size of the partition is then reduced by this probability and the item is removed from the set of items to be partitioned. Next if any space remains in the partition, the item with largest edge weight probability is placed into the partition with probability proportional to the remaining space in the partition, the item's probability is then reduced by this amount and it is added back to the set of items to be partitioned provided

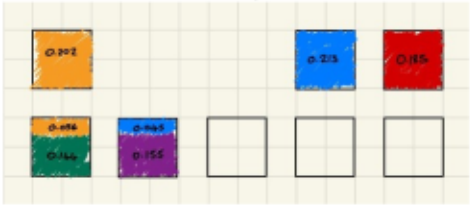
its probability has not been reduced to zero. Items are only considered while their probabilities remain above zero and throughout the process the items with smallest and largest probabilities will change due to the reduction of probabilities after a partitioning step. Once all items have been partitioned a uniform random number is generated, this number is used to select a partition and also a portion of that partition by dividing the number by n. The portion of the partition selected represents which item, or in this case which edge, has been sampled. The entire process of partitioning and sampling is shown in figure 5.9. The alias method provides very efficient sampling due to the fact that pseudo-random uniform numbers may be generated incredibly efficiently, as well as this the partitioning process as a pre-processing step may be carried out in $O(n)$ providing probabilities can be calculated in constant time.



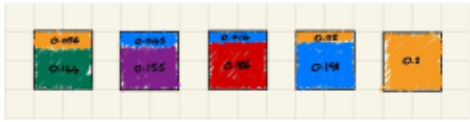
(a) The edge weight probabilities (top) and 5 empty partitions (bottom) of size $\frac{1}{5}$ are shown.



(b) The smallest probability fills the first partition, the largest fills the remaining space.

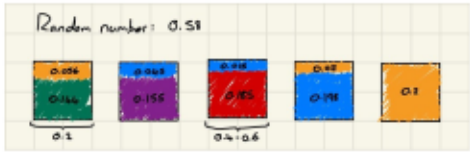


(c) The process shown in 5.9b is repeated over all partitions.

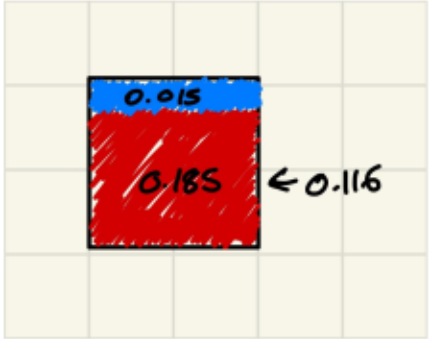


(d) Finally, each partition contains portions of the original probabilities.

(e) A random number between 0 and 1 is generated as 0.58.



(f) The third bin is selected as the random number lies in its interval.



(g) The random number is divided by $n = 5$ and lies within the red portion, therefore it is selected.

Figure 5.9: *Edges in the neighbourhood of E being partitioned based on edge weight probability, then sampled from in order to select an edge. Adapted from a diagram by Stefek (2017).*

6 Implementation Details

This section gives a detailed outline of the implementations of the algorithms and data structures used throughout the project. Such details include data structures and external algorithms that allow each compression algorithm to function correctly and efficiently, and the compression algorithms themselves. Java was used to implement all algorithms and data structures. Each sparsifier may be viewed as a method to remove a single vertex from a graph, and to compress a graph via some method is to use this method alone to remove all non-terminals. This alteration allows the development of hybrid approaches which combine multiple sparsifiers, as discussed in section 6.2.9.

6.1 Graph Components

6.1.1 Graph Data Structure

The graph data structure used in this project simply maintains an indexable list of vertex objects which store most important data about the graph. The graph maintains several utility functions for calculating the size of the graph either before or after compression, as well as an implementation of Dijkstra's shortest path algorithm (Dijkstra (1959)) which makes use of a min-priority queue.

6.1.2 Vertex Data Structure

The vertex data structure maintains many core attributes of the graph, relevant time complexities for the algorithms discussed here have been detailed in section 4. It consists of an index allowing for it to be located within the graph's vertex list, and an adjacency list represented by a Java hashmap where each entry is a vertex index, edge weight key-value pair used to represent edges within the graph. Utility fields used in calculating shortest paths include Boolean values indicating whether or not the vertex has been visited or is contained in the priority queue, a float representing the shortest path from some starting point to the vertex, and a pointer to the previous, or parent, vertex in the shortest path which is null where no shortest path has been calculated. The vertex data structure also maintains two Boolean values used in graph compression indicating whether or not the vertex is a terminal, and also whether or not it has been deactivated which indicates that the vertex has been removed from the graph.

This data structure is also responsible for the partitioning and random sampling of edges to contract when using the REC algorithm. It contains a Boolean field indicating whether or not the vertex has been partitioned and a list of partitions like those shown in figure 5.9. Each partition itself is represented as a 3-tuple consisting of the once smallest edge weight probability during the steps carried out in figure 5.9, the corresponding index of the adjacent vertex incident to this edge, and the index of the adjacent vertex whose probability was used to fill the remainder of the partition. The process carried out is identical to the process shown in figure 5.9 where each partition is represented as a 3-tuple as described in section 6.3. Algorithm 1 shows the partition function.

Data: The neighbourhood, $N = (V, E, c)$, of some vertex $v \in V$.
Result: The partitioning of edges of a vertex v.
begin
  $t \longleftarrow$ an empty AVL-Tree
  for *each edge in the adjacency list of* v **do**
    $p \longleftarrow$ the probability of the edge
    $i \longleftarrow$ the index of the adjacent vertex
    $t.insert(key = p, value = i)$
  end
  $n \longleftarrow |E|$
  create n partitions each of size $1/n$
  for *each partition* b **do**
    $p_{smallest}, i_{smallest} \longleftarrow t.popmin()$
    $p_{largest}, i_{largest} \longleftarrow t.popmax()$
    $b.size \longleftarrow b.size - p_{smallest}$
    $p_{largest} \longleftarrow p_{largest} - b.size$
    if $p_{largest}$ *is not zero* **then**
      $t.insert(key = p_{largest}, value = i_{largest})$
    end
    $b.tuple = (p_{smallest}, i_{smallest}, i_{largest})$
  end
end

Algorithm 1: The partition function.

An AVL–Tree, a self-balancing binary search tree as described by AdelsonVelskii and Landis (1963), is used to store and retrieve edge weight probabilities and their corresponding adjacent vertices during partitioning. An AVL-tree is an efficient data structure as the self-balancing nature of the tree allows for the quick retrieval of the smallest and largest key values contained within the tree. The Vertex data structure also handles the sampling process and algorithm 2 shows the details of this function. Here a random partition is selected from the list of partitions and a random float is generated using Java's Math.random() on the interval $[0, 1/|p|]$ where $|p|$ is the number of partitions or alternatively the number of edges that the vertex is incident to. This float is used to determine which of the two probabilities, and therefore which of the two edges to select for contraction, where if the float falls below the value of the smaller probability its corresponding edge is chosen, and otherwise the other edge is chosen .

Data: A vertex v
Result: The index of the vertex adjacent to v which v will be merged with via edge contraction.
begin
  $v.partition()$
  $p \longleftarrow v.getPartitions()$
  $b \longleftarrow$ a partition selected randomly from p
  $r \longleftarrow$ a random float on $[0, 1/|p|]$
  if $r \leq b.p_{smallest}$ **then**
    return $i_{smallest}$
  else
    return $i_{largest}$
end

Algorithm 2: The partition sampling process.

6.2 Sparsifier Class

The sparsifier class is responsible for maintaining all sparsification algorithms and several utilities such as generating and assigning sets of terminals described in section 6.2.1. A sparsifier is created by passing it a graph object as described in section 6.1. The data structure maintains a string value representing which sparsification algorithm should be used, an integer representing the percentage of non-terminals to contract before halting in the early stopping variation of REC described in section 6.2.8, a list of terminals represented as vertex indexes, and a Boolean value indicating whether or not to use the independent set variation of REC described in section 6.2.7.

6.2.1 Terminal Generation

The sparsifier maintains a list of terminal vertices within the given graph which is used to determine which vertices to remove during compression. These may be set manually or generated randomly using a utility function within the class. Terminal sets may also be extracted and shared between different sparsifiers in order to compare experiment results on the same terminal sets.

6.2.2 Minimum Degree Heuristic

Each sparsifier makes use of the minimum degree heuristic in order to determine the order in which to remove non-terminals from the graph. This is considered a desirable approach as removing vertices with lower degree initially may reduce the negative effects of approximate compression as fewer edges will be effected. The degree of a vertex is simply the number of edges that the vertex is incident to, and each compression algorithm aims to remove non-terminals from a graph in ascending order of their degree. Prior to compression, all non-terminals are stored in a priority queue where the priority of a vertex is its degree pre-compression, the population of this queue has worst case time complexity $O(n)$. This allows for the constant time retrieval of the non-terminal with minimum degree each time a non-terminal is to be removed from the graph. While this approach is efficient, throughout compression the degree of vertices may change due to the deletion of edges within the graph, and therefore vertices are only removed in order of their initial degree before compression has begun.

6.2.3 Quality Assessment

The sparsifiers detailed in this project are known as distance preserving sparsifiers and aim to preserve shortest path distances during compression. Quality calculations are carried out as described in section 4.4. Prior to compression, each terminal is used as a starting position for calculating the shortest path to every other terminal in the graph and these distances are stored as a matrix where each row and column represent each terminal. The values stored in the matrix are the shortest path distances between the corresponding terminals represented by the row and column. After the graph has been compressed, shortest paths are calculated again and the shortest path matrix becomes a quality matrix by dividing the values in the compressed graph quality matrix by the values in the uncompressed graph quality matrix. The result of this process is a set of quality values which indicates how much longer each shortest path is in the compressed graph.

Due to the requirement of computing many shortest paths, the quality assessment process is computationally expensive especially in larger graphs. The quality calculations require an all pairs shortest paths computation to be carried out over the set of terminals in the graph. This may be carried out in two ways, either using the Floyd-Warshall algorithm (Floyd (1962)) which computes all pairs of shortest paths in a graph with a time complexity of $O(n^3)$ for $n = |V|$, or using Dijkstra's shortest path algorithm (Dijkstra (1959)) using each terminal as a starting position to construct a shortest path tree with a time complexity of $O(k \cdot (n + m \log n))$ where

the algorithm is implemented using a min-priority queue, $k = |T|$, $n = |V|$, and $m = |E|$. This project employs the latter approach using Dijkstra's shortest path algorithm in order to construct the quality matrix.

Remark 1 *Where computational complexity is concerned, it seems reasonable to assert that assessing the quality of a sparsifier is a more difficult task than graph sparsification itself.*

6.2.4 Gaussian Elimination Compression Algorithm

```
Data: A graph G = (V, E, c) and a non-terminal vertex v ∈ V.
Result: The removal of the vertex v from V and edges it is
        incident to from E.
begin
    for each neighbour u1 of v do
        for each neighbour u2 of v do
            if u1 ≠ u2 then
                x ←− c(v, u1) + c(v, u2)
                if u1 and u2 are adjacent then
                    c(u1, u2) = min{c(u1, u2), x}
                else
                    create new edge (u1, u2) ∈ E with
                     c(u1, u2) = x
            end
        end
        remove the edge (v, u1) from E
    end
    v.deactivate()
end
```

Algorithm 3: The Gaussian elimination algorithm.

The Gaussian elimination algorithm, shown in algorithm 3, does not use any external algorithms and functions iteratively over the set of non-terminals. For each non-terminal, the algorithm will iterate over each of its neighbours and compare it to every other one of its neighbours in a nested iteration. At each nested iteration if no edge exists between the two neighbours then one will be created with the weight of the path between the neighbours through the vertex being removed, if an edge exists then its weight will either remain unchanged or be updated if the path through the vertex being removed is shorter. During this process all edges incident the vertex being removed will be deleted and the vertex will be marked as deactivated indicating that is has been removed from the graph. As briefly mentioned in section 5.1 this algorithm has a worst-case time complexity of $O(m \cdot n^2)$ for a fully connected graph $G = (V, E, c)$ where $m = |V| - |T|$ and $n = |V|$. The main body of the algorithm requires iteration over each neighbour of v, and a further nested iteration over these neighbours, resulting in a time complexity of $O(n^2)$, and this process is performed for each non-terminal, of which there are $|V| - |T|$, resulting in the overall time complexity shown.

6.2.5 Shortest Path Tree Compression Algorithm

The shortest path tree method is implemented as an adaptation of a description by Krauthgamer and Zondiner (2012) and makes use of Dijkstra's shortest path algorithm (Dijkstra (1959)) for calculating the shortest paths starting from each terminal to every other terminal in the graph. This constructs a tree of shortest paths which can be back-tracked from all terminals to the starting terminal which is itself a graph minor. A complete shortest path tree is constructed as the union of all shortest path trees starting from each terminal to every other terminal in the graph where lower edge weights are favoured during the union of shortest path trees. A form of edge contraction is carried out on this shortest path tree where each non-terminal remaining in the tree is replaced by an edge between its two neighbours. The resulting graph is an incomplete compression as some non-terminals may remain as discussed in section 5.2. Algorithm 4 Shows the process carried out by this compression method, as Dijkstra's shortest path algorithm (Dijkstra (1959)) is used for constructing a shortest path tree starting from each terminal, the time complexity is $O(k \cdot (n + m \log n))$ where Dijkstra's algorithm is implemented using a min-priority queue, $k = |T|$, $n = |V|$, and $m = |E|$ as in quality assessment.

Data: A graph $G = (V, E, c)$ and a terminal set T
Result: The potentially incomplete compression of the graph G.
begin
  for *each terminal* $t \in T$ **do**
    construct the shortest path tree starting from t
  end
  construct $G_T = (V_T, E_T, c)$ as the union of all shortest path trees
  deactivate all non-terminals not in V_T
  for *each non-terminal* $n \in V_T$ **do**
    if $n.degree() == 2$ **then**
      add edge (u_1, u_2) to E_T with $c(u_1, u_2) = c(u_1, n) + c(u_2, n)$ where u_1 and u_2 are the two neighbours of n
      remove edges $(u1, n)$ and (u_2, n) from E_T
      $n.deactivate()$
    end
  end
end

Algorithm 4: The shortest path tree algorithm adapted from an algorithm described by Krauthgamer and Zondiner (2012).

6.2.6 Random Edge Contractions Compression Algorithm

The random edge contractions algorithm functions as detailed in section 5.3. The algorithm partitions and samples edges to contract using the alias method (Kronmal and Peterson Jr (1979)) for random sampling. For each non-terminal an adjacent vertex, referred to as a supernode, is selected, the weight of the edge connecting the two is stored and the edge is removed from the graph, after this each of the non-terminal's neighbours is considered. For each neighbour of the non-terminal the edge connecting the two is removed from the graph and a new edge is created between the neighbour and the supernode with the weight of the two removed edges. Where an edge between the neighbour and the supernode already exists the weight is updated to be the lower of either the sum of the two removed edges, or the original weight of the edge. The REC algorithm is shown in algorithm 5. The time complexity of the partitioning and sampling

process has been discussed previously in section 4. For a given vertex, v, in a graph $G = (V, E, c)$, the algorithm functions on the neighbourhood, $N = (V', E', c)$, of v and iterates over the edges v is incident to only once, giving an overall time complexity of $O(n)$ for $n = |E'|$ providing all operations may be performed in constant time.

Data: A graph $G = (V, E, c)$ and a non-terminal vertex $v \in V$.
Result: The removal of the vertex v from V and edges it is incident to from E.

```
begin
    u ⟵ V.get(v.sample())
    x ⟵ c(u, v)
    remove (u, v) from E
    for each neighbour w of v do
        if (w, u) ∈ E then
            c(w, u) = min{c(w, u), x + c(w, v)}
        else
            create new edge (w, u) in E with c(w, u) = x + c(w, v)
        remove (w, v) from E
    end
    v.deactivate()
end
```

Algorithm 5: The REC algorithm.

An improvement to the efficiency of this algorithm is to handle vertices of degree ≤ 2 prior to partitioning and sampling. This improvement was inspired by the SPT algorithm in which non-terminals of degree 2 are replaced by an edge between their neighbours with the weight of the sum of the two edges incident to the non-terminal. Unlike the SPT algorithm the REC algorithm may encounter non-terminals of degree 1, these and their single edges may simply be removed from the graph as the sampling of a single edge is inefficient and illogical. This improvement may also contribute to improved quality where the graph contains many non-terminals of degree 2 due to reduced impact on edge weights.

6.2.7 Random Edge Contractions with Independent Set

A variation of the REC sparsifier is to contract only the non-terminals contained within an independent set of vertices of the graph. An independent set is a subset of vertices where there are no edges in the graph between vertices in the independent set. The maximum independent set problem is NP-Hard and therefore it must be approximated using some approximation algorithm. The approach outlined here makes use of an approximate algorithm as detailed by Chekuri (2021). While there is some non-terminal remaining in the graph the largest approximate independent set is constructed by taking the vertex with smallest degree and adding it to the set, then excluding all of its neighbours from being included. This process is repeated until an independent set is formed and these non-terminals are then removed from the graph. As these independent sets only contain non-terminals they will differ after each independent set has been removed from the graph, and therefore this process may be repeated until no more non-terminals remain in the graph. It is possible to remove all non-terminals from the graph as there will always be an independent set of size 1, however if too many independent sets of size 1 exist then this algorithm will be no different than standard REC. This approach is treated as a separate algorithm from standard REC and will be referred to as RECIndSet. The independent set approximation algorithm is shown in algorithm 6.

Data: A graph $G = (V, E, c)$ and a set of non-terminals T
Result: The approximated largest independent set of non-terminals.
begin
  $N \longleftarrow$ the set of non-terminals $V - T$
  $S \longleftarrow \emptyset$
  while $N \neq \emptyset$ **do**
    add non-terminal v with minimum degree from N to S
    remove v and its neighbours from N
  end
  return S
end

Algorithm 6: The approximate independent set algorithm adapted from a description by Chekuri (2021).

6.2.8 Random Edge Contractions with Early Stopping

Another variation of the REC algorithm is to allow the algorithm to halt prematurely after a certain percentage of non-terminals have been removed from the graph. In this approach, the REC algorithm is given a percentage, for example 75, and will only remove the corresponding number of non-terminals from the graph using the minimum degree heuristic. This, like the SPT method, is an example of an incomplete compression, where some non-terminals remain in the graph after compression. Like RECIndSet, this variation is treated as a separate algorithm and will be referred to as REC followed by the percentage of non-terminals to be contracted, for example for 75 percent the variation will be referred to as REC75.

6.2.9 Random Selection Compression Algorithm

The final alternative variation of REC considered is a hybrid approach which combines the approximate REC method with the exact Gaussian elimination method. In this approach, when a non-terminal is selected for removal from the graph the algorithm will randomly choose to remove it with either the REC or Gaussian elimination algorithm. This variant will be referred to as the random selection algorithm or Rand.

6.3 Other Utilities

Other utilities used throughout this project are implementations of already mentioned data structures, these being a 3-tuple, a min-priority queue, and an AVL-Tree. The-AVL tree is implemented as described by AdelsonVelskii and Landis (1963) with insertion and retrieval of maximum and minimum values having a time complexity of $O(\log n)$. The min-priority queue is based on an underlying linked list data structure, therefore the worst case insertion time complexity is $O(n)$ and retrieval is $O(1)$. The 3-tuple data structure is simply a generic data structure that stores three values, this is used to represent partitions in the alias method.

7 Experimental Evaluation

This section details the experiments carried out with each of the previously described sparsifiers. Each sparsifier was used to compress different graphs of various sizes, and the worst and average quality values and the run times are reported and compared. As the approximate sparsifier and its variations make use of random elements each sparsifier will compress each graph ten times, the set of quality values, as described in section 6.2.3, and run times will be averaged. Final run times are reported as this average and average and worst qualities are extracted from this averaged quality set.

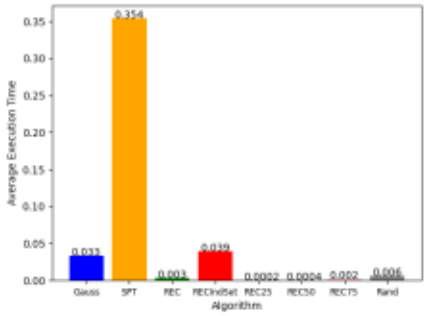
The exact sparsifiers, Gauss and SPT, will be used as a baseline for comparisons of several variants of the approximate REC method, these variants include REC, REC with early stopping at 25, 50, and 75 percent of non-terminals, RECIndSet, and random selection. The experiments have been carried out on a personal machine with an 8-core Intel i5 2.5Ghz processor and 32GB RAM. An additional experiment for the early stopping variation of REC was carried out where each percentage from 1 to 100 was given to the algorithm in order to determine an optimal halting percentage, if one exists.

7.1 SNAP Graphs

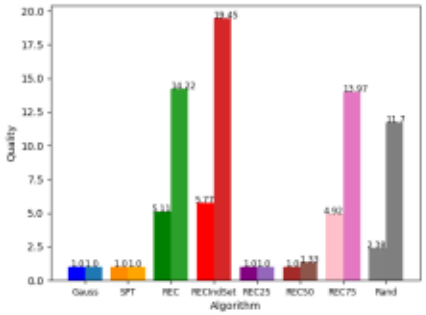
The graphs used to evaluate each algorithm are taken from the Stanford Network Analysis Platform (SNAP) (Leskovec and Krevl (2014)). SNAP provides a network analysis and graph library as well as a collection of many graphs. Two smaller graphs have been selected from the University of Oregon's RouteViews project. The aim of the project was to construct graphs based on autonomous systems using daily reports over the course of 785 days beginning on the 8th of November 1997. Over this period of time, 733 graphs of autonomous systems were constructed. The dataset has been used to explore how graphs evolve over time (Leskovec et al. (2005)). This project considers two of these graphs, one from the 8th of November 1997 consisting of 3015 vertices and 5347 edges, referred to as 3K AS, and one from the 2nd of January 2000 consisting of 6474 vertices and 13233 edges, referred to as 6K AS. Large graphs appear frequently in social media analysis, and therefore it seems appropriate to consider how the compression algorithms perform in this area. Two location-based social media platforms, BrightKite and Gowalla, had friendship graphs constructed in order to explore the impact of social media on human movement (Cho et al. (2011)). This project aims to compress both of these graphs consisting of 58228 vertices and 214078 edges for BrightKite, referred to as 60K BrightKite, and 196591 vertices and 950327 edges for Gowalla, referred to as 200K Gowalla. As the autonomous systems graphs are relatively small a randomly generated terminal set of size 40 will be used for each compression. A terminal set of size 10 will be used for the larger social network graphs as the quality assessment phase proves to be quite time consuming when calculating shortest paths for larger terminal sets in large graphs, this is discussed further in section 6.2.3. For a given graph all algorithms employ the same terminal set in order to ensure fair comparison between compressions.

7.2 Autonomous System Graphs

7.2.1 3K AS Graph



(a) The average execution times of each compression algorithm.



(b) The average and worst qualities of each compression algorithm, average quality is shown on the left bar and worst on the right.

Figure 7.1: *Results observed when compressing the 3K AS graph.*

Figure 7.1 shows the results of compressing the graph with each algorithm. It can be seen that in such a small graph all algorithms perform incredibly efficiently, and the Gaussian elimination method may compress the graph with perfect quality in a fraction of a second. Interestingly RECIndSet appears to show worse quality than standard REC, and the random selection algorithm shows better quality perhaps due to the combination of exact compression. The early stopping variant REC50 appears to provide incredibly efficient near perfect compression, however 50 percent of non-terminals will remain in the compressed graph and it may still be desirable to employ the Gaussian elimination method in such a small graph. Figure 7.2 supports the choice of REC50 as it can be seen that quality soon becomes unmanageable after more than 50 percent of non-terminals have been contracted. We can also see that in this instance the SPT method appears to be the slowest method for compressing the given graph, however later results show that this is due to the size of the graph failing to highlight the inefficiency of other compression methods.

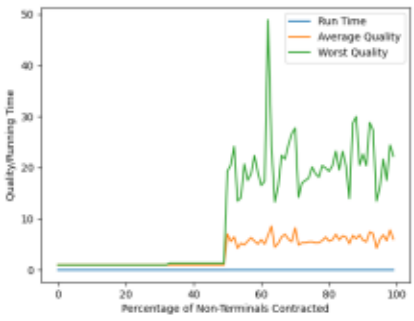
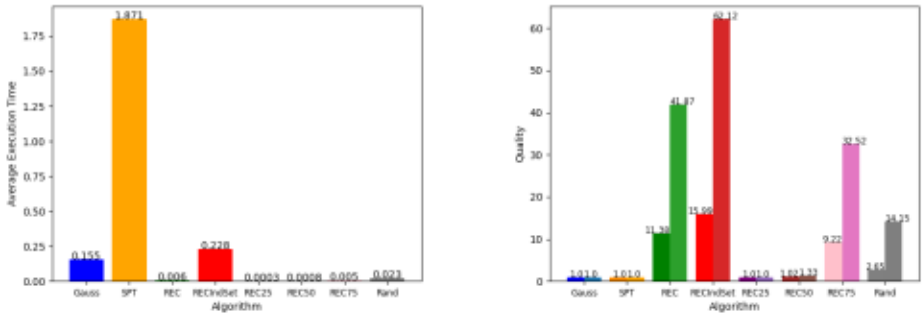


Figure 7.2: *Results observed when compressing the 3K AS graph at each percentage of early stopping from 1 to 100.*

7.2.2 6K AS Graph



(a) The average execution times of each compression algorithm.

(b) The average and worst qualities of each compression algorithm, average quality is shown on the left bar and worst on the right.

Figure 7.3: *Results observed when compressing the 6K AS graph.*

The graphs in figure 7.3 show the results observed when compressing the 6K AS graph with each sparsifier. The results appear to echo the discussion relating to the 3K AS graph in that, while all algorithms are efficient at this size, it is desirable to employ the Gaussian elimination exact sparsifier for perfect compression. Again, the SPT method is by far the slowest. It is here where the emergence of graph size directly impacting the quality of the REC sparsifier begins to show. Comparing the results in figures 7.1b and 7.3b shows a drastic increase in worst and average quality values when compressing a larger graph. This behaviour is also observed for RECIndSet and REC75, but not for Rand and this may be due to the combination of exact and approximate compression. In the case of REC75, this could imply that there exists some graph or terminal set size for a given graph that must not be exceeded before quality becomes unmanageable. This is supported by figures 7.2 and 7.4 which appear to indicate that fifty percent is the optimal early halting procedure for the two graphs shown thus far.

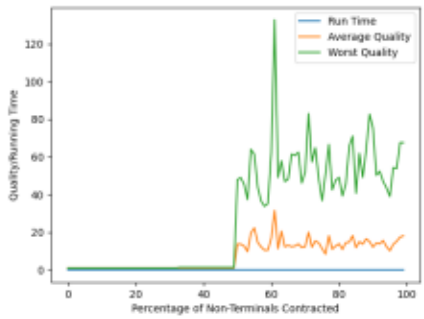
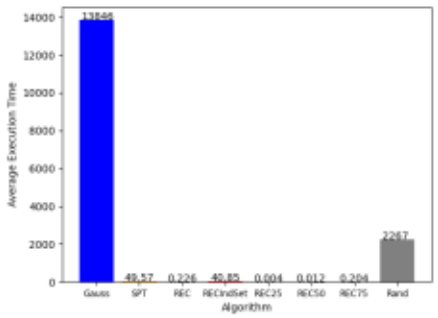


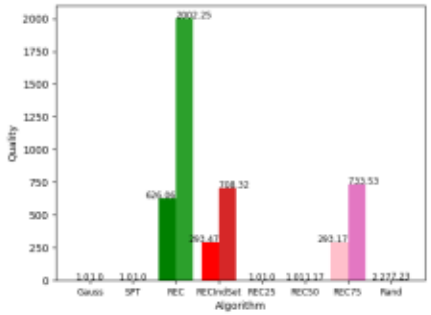
Figure 7.4: *Results observed when compressing the 6K AS graph at each percentage of early stopping from* 1 *to* 100.

7.3 Social Network Graphs

7.3.1 60K BrightKite Graph



(a) The average execution times of each compression algorithm.



(b) The average and worst qualities of each compression algorithm, average quality is shown on the left bar and worst on the right.

Figure 7.5: *Results observed when compressing the 60K BrightKite graph.*

Figure 7.5 shows the results of compressing the 60K BrightKite graph. One thing that immediately becomes clear is that the Gaussian elimination method is not a feasible approach for such a large graph, and as a result the Rand sparsifier also becomes redundant and both are seen to have incredibly large run times. SPT is no longer the worst performing sparsifier, however it is also an inefficient approach with a relatively large run time for the graph. RECIndSet has a similar run time to SPT and therefore is also considered an inefficient approach. Figure 7.5b also supports earlier comments relating to graph size directly impacting the quality of a sparsifier, and a huge increase in quality values can be seen for REC, RECIndSet, and REC75. While Rand is incredibly inefficient at this graph size, its quality still remains very consistent. At this point, REC25 and REC50 appear to be the most practical sparsifiers and, although they do not provide complete compression, the run times are very efficient and they provide near-exact quality compression. Figure 7.6 also serves to solidify the previous statement that early halting after contracting fifty percent of non-terminals is a potentially optimal procedure for efficient, incomplete compression.

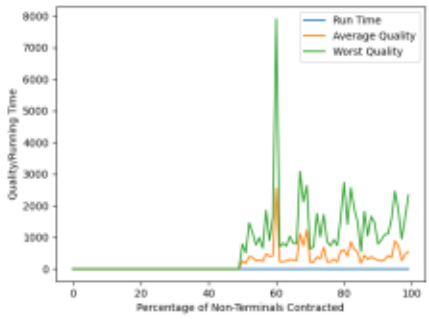
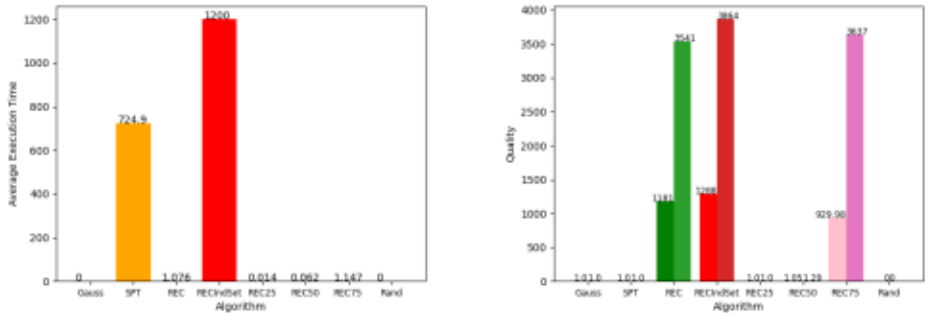


Figure 7.6: *Results observed when compressing the 60K BrightKite graph at each percentage of early stopping from 1 to 100.*

7.3.2 200K Gowalla Graph



(a) The average execution times of each compression algorithm.

(b) The average and worst qualities of each compression algorithm, average quality is shown on the left bar and worst on the right.

Figure 7.7: *Results observed when compressing the 200K Gowalla graph.*

Finally, figure 7.7 shows the results of compressing the 200K Gowalla graph. Due to memory limitations compression was not possible using the Gaussian elimination sparsifier, and as a result the Rand sparsifier also, therefore they have not been included in figure 7.7a. Rand has also been omitted from figure 7.7b, but Gauss has not as its quality will always be 1.

The results seen here serve to reinforce previous statements made about all algorithms. The quality values of REC variants are considerably affected by the size of the graph, and therefore are not considered worthwhile although run times remain fast. RECIndSet and SPT also display similar behaviour to that in previous graphs, where run times become larger with graph size. REC25 and REC50 are the only remaining approaches considered feasible, with incredibly low run times and near-perfect quality compression. This statement is further cemented by the results shown in figure 7.8, which shows that even in such a large graph fifty percent remains the optimal early halting procedure.

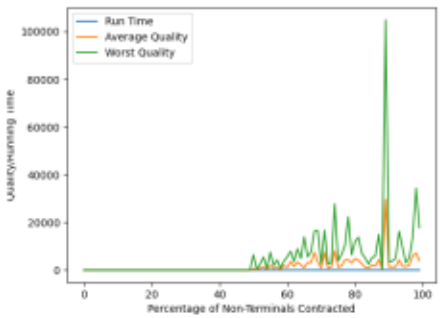


Figure 7.8: *Results observed when compressing the 200K Gowalla graph at each percentage of early stopping from 1 to 100.*

8 Conclusion

The results in section 7 show that, while vastly out-performing the exact sparsifiers in terms of running times, the size of a graph directly affects the quality of the REC sparsifier, where larger graphs imply worse quality. This behaviour is not reflected in either the REC25 nor the REC50 variants which can be seen to provide near-exact sparsification even in the largest graph. Figures 7.2, 7.4, 7.6, and 7.8 show that halting after contracting fifty percent of non-terminals appears to be the optimal policy, and quality becomes unmanageable after this point. It is unclear whether or not this algorithm behaves similarly to REC, in that larger graphs reduce the quality, and further experimentation on such graphs may show that REC50 becomes ineffective at a certain size threshold. If such a size exists, further investigation may motivate the development of a more effective sparsifier. Gaussian elimination remains optimal for smaller graphs where exact compression may be achieved in a fraction of a second, however the algorithm becomes impractical for larger graphs due to long run-times. The random selection algorithm suffers from this same inefficiency in larger graphs, however the quality of the sparsifier appears to remain incredibly consistent even in larger graphs. As this algorithm randomly selects between sparsification techniques blindly, further investigation into the selection process could prove useful in providing better results and faster run times. A potential improvement may rely on a machine learning approach where sparsifiers are selected randomly from a defined library of algorithms. A similar approach is SATZilla (Xu et al. (2008)), an algorithm selection tool for solving the NP-Complete Boolean Satisfiability Problem. As mentioned in section 6.2.3, quality assessment is much more time consuming than the compression itself. Investigation into carrying out quality assessments more efficiently would allow for much more thorough experimentation, especially for larger graphs. This is also one limitation identified during the project, as the computational cost of performing quality checks on large graphs was either not possible or incredibly time-consuming.

8 Bibliography

M. AdelsonVelskii and E. M. Landis. An algorithm for the organization of information. Technical report, JOINT PUBLICATIONS RESEARCH SERVICE WASHINGTON DC, 1963.

A. A. Benczúr and D. R. Karger. Approximating st minimum cuts in õ (n 2) time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 47–55, 1996.

C. Chekuri. Cs 583: Aproximation algorithms, November 2021. URL: "`https://courses.engr.illinois.edu/cs583/sp2016/LectureNotes/packing.pdf`".

Y. K. Cheung, G. Goranci, and M. Henzinger. Graph minors for preserving terminal distances approximately-lower and upper bounds. *arXiv preprint arXiv:1604.08342*, 2016.

E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 1082–1090, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308137. doi: 10.1145/2020408.2020579. URL `https://doi.org/10.1145/2020408.2020579`.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1): 269–271, 1959.

T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

A. Gupta. Steiner points in tree metrics don't (really) help. In *SODA*, volume 1, pages 220–227, 2001.

L. Kamma, R. Krauthgamer, and H. L. Nguyn. Cutting corners cheaply, or how to remove steiner points. *SIAM Journal on Computing*, 44(4):975–995, 2015.

D. R. Karger. Using randomized sparsification to approximate minimum cuts. In *SODA*, volume 94, pages 424–432, 1994.

D. R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000.

R. Krauthgamer and T. Zondiner. Preserving terminal distances using minors. 2012.

R. A. Kronmal and A. V. Peterson Jr. On the alias method for generating random variables from a discrete distribution. *The American Statistician*, 33(4):214–218, 1979.

R. Kyng and S. Sachdeva. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 573–582. IEEE, 2016.

J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, page 177–187, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 159593135X. doi: 10.1145/1081870.1081893. URL `https://doi.org/10.1145/1081870.1081893`.

A. Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 3–12. IEEE, 2009.

H. Räcke, C. Shah, and H. Täubig. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 227–238. SIAM, 2014.

D. J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970.

D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances, 2008. URL `https://arxiv.org/abs/0803.0929`.

D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40 (4):981–1025, 2011.

P. Stefek. The alias method for fast weighted random sampling. `https://www.peterstefek.me/alias-method.html`, 2017.

R. E. Tarjan. *Graph Theory and Gaussian Elimination.* PhD thesis, Stanford University, 1975.

L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.